Asymmetric Memory Fences: Optimizing Both Performance and Implementability *

Yuelu Duan, Nima Honarmand, † and Josep Torrellas
University of Illinois at Urbana-Champaign
{duan11,torrella}@illinois.edu nhonarmand@cs.stonybrook.edu
http://iacoma.cs.uiuc.edu

Abstract

There have been several recent efforts to improve the performance of fences. The most aggressive designs allow post-fence accesses to retire and complete before the fence completes. Unfortunately, such designs present implementation difficulties due to their reliance on global state and structures.

This paper's goal is to optimize both the performance and the implementability of fences. We start-off with a design like the most aggressive ones but without the global state. We call it *Weak Fence* or *wF*. Since the concurrent execution of multiple *wF*s can deadlock, we combine *wF*s with a conventional fence (i.e., *Strong Fence* or *sF*) for the less performance-critical thread(s). We call the result an *Asymmetric* fence group. We also propose a taxonomy of Asymmetric fence groups under TSO. Compared to past aggressive fences, Asymmetric fence groups both are substantially easier to implement and have higher average performance. The two main designs presented (*WS+* and *W+*) speed-up workloads under TSO by an average of 13% and 21%, respectively, over conventional fences.

Categories and Subject Descriptors C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - Multiple-instruction-stream, multiple-data-stream processors (MIMD); D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel programming.

Keywords Fences; Sequential Consistency; Synchronization; Parallel Programming; Shared-Memory Machines.

1. Introduction

Fence instructions prevent the compiler and the hardware from reordering memory accesses [13, 32]. In its basic form, a fence instruction prevents post-fence accesses from being observed by other processors before all pre-fence accesses have completed.

Programmers and compilers use fence instructions to coordinate threads with low overhead — where more popular synchronization primitives would be too heavyweight. For example, programmers insert fences in performance-critical codes with fine-grain communication. Examples include runtime systems such as Cilk [9] and Threading Building Blocks [27], synchronization libraries, operating systems, and Software Transactional Memory (STM) systems [30]. Compilers also insert fences, e.g., when generating code for accesses to variables declared atomic in C++ (or volatile in Java). Often, fences appear in performance-critical codes.

Successive generations of processors have steadily reduced fence overhead. One useful technique is to execute post-fence loads speculatively within the reorder buffer before the fence completes — i.e., before all the pre-fence loads retire and all the pre-fence stores drain from the write buffer. A post-fence load only stalls when it is about to retire. If, before the fence completes, an external coherence message conflicts with a speculative load, the load is retried.

In practice, however, a fence is often costly, especially if the write buffer is full with several pre-fence stores that miss in the cache, and the memory consistency model requires draining stores one at a time, such as in TSO. Recently, we measured the stall of a fence preceded by many writes in an 8-threaded Intel Xeon E5530 desktop to be ≈200 cycles [8].

As a result, there have been several recent proposals of high-performance fence designs (e.g., [8, 15, 19, 20]). The most aggressive of these are WeeFence [8] and Address-Aware Fences (AAF) [19], which allow post-fence accesses to *retire and complete* before the fence completes.

However, these two schemes present implementation difficulties. When a processor encounters a fence, it needs to obtain some global state. In a distributed-directory environment, with multiple processors updating and reading this global state, obtaining a consistent view is very challenging — in fact, we believe that the problem is still unsolved. In addition, supporting the global state requires non-trivially augmenting protocol messages and adding hardware structures. It would be most helpful to have a fence with the access reordering abilities of WeeFence or AAF, but *without* any global state. We call such a design a *Weak Fence (wF)*.

Fences prevent a Sequential Consistency Violation (SCV) when multiple fences execute concurrently, each one invoked by a different thread and, as a group, prevent a cycle of dependences [29]. We call an instance of these dynamic groups a *Fence Group*. Unfortunately, we find that, if all the fences in

* This work was supported in part by NSF under grants CCF-1012759 and CNS-1116237, and Intel under the Illinois-Intel Parallelism Center (I2PC).

† Nima Honarmand is now with the Department of Computer Science, Stony Brook University.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.

http://dx.doi.org/10.1145/2694344.2694388

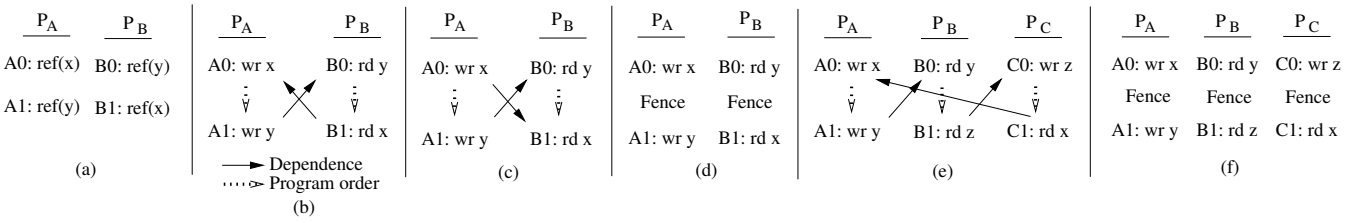

Figure 1. Examples of SC violations and uses of fences.

a group are *wF*s, when they prevent an SCV, they deadlock. Hence, this design is not acceptable. However, a key insight of this paper is that, if at least one of the fences in a group is a conventional one, then there can be no deadlock or SCV. We call a conventional fence a *Strong Fence (sF)*, and a fence group that contains one or more *sF*s and one or more *wF*s an *Asymmetric* fence group. Implementing Asymmetric groups is much simpler than implementing WeeFences or AAFs.

Interestingly, a fence group often contains one or more threads that are more performance-critical than the other(s). This idea was pointed out by Ladan-Mozes et al. [15] for two-fence groups. Hence, we use *wF*s in the critical threads and *sF*s in the other threads. For example, in the Cilk runtime [9], a thread accessing its own task queue is the common case and can use a *wF*, while a second thread stealing a task from the first thread's task queue is rarer, and can use an *sF*. With this strategy, Asymmetric fences perform comparably to or *even better* than WeeFences.

Based on these insights, this paper proposes *Asymmetric Fences*, which provide the best fence performance-cost tradeoff that we are aware of. It also proposes a *taxonomy* of Asymmetric fence groups under TSO, with various designs optimized for different needs. While Asymmetric fences introduce interesting programming issues, in this paper, we do not address such issues, and leave them for future work. However, we expect Asymmetric fences to be used by expert programmers, as they write performance-critical code.

Overall, the main contributions of this paper are:

- The proposal of Asymmetric fences, which combine *sF*s and *wF*s in the same fence group.
- A taxonomy of Asymmetric fence groups under TSO.
- The description of a few uses of Asymmetric fences.
- An evaluation of the performance, characteristics, and scalability of Asymmetric fences. The average performance of the two main designs (*WS+* and *W+*) under TSO is higher than past aggressive fences, and is 13% and 21% higher, respectively, than conventional fences.

In this paper, Section 2 gives a background; Section 3 presents Asymmetric fences; Section 4 discusses several example uses; Section 5 discusses hardware and programming issues; Sections 6-7 evaluate Asymmetric fences; and Section 8 covers related work.

2. Motivation

2.1 Background on Fences

To understand the use of fences, we begin by defining *performed*, *retired*, and *completed* for a memory instruction. A load *performs* when the data loaded returns from the memory system and is deposited into a register. It *retires* when it reaches the head of the Reorder Buffer (ROB) and has performed. After retirement, the load has *completed*.

A store *retires* when it reaches the head of the ROB and its address and data are available. The store goes into the write buffer. Later, when the memory consistency model allows, the store merges with the memory system, potentially triggering a coherence transaction. When the latter terminates (e.g., when all the invalidation acknowledgments have been received), the store has *performed*, and is now *completed*. Then, the store is removed from the write buffer.

The memory consistency model determines the access reorderings allowed. TSO [32] allows reodering between a store and a subsequent load, but not between two loads or between two stores. Release Consistency (RC) [10] allows all these three reorderings. In the write buffer, TSO only allows one write to merge with the memory system at a time, while RC allows multiple writes to merge concurrently.

Fences are instructions that prevent the compiler and the hardware from reordering memory accesses [13, 32]. While there are different flavors of fences, the basic idea is that: (1) a fence *completes* when all pre-fence accesses have *completed*, and (2) a fence has to complete before any post-fence access can be observed by other processors. Of course, post-fence loads can execute speculatively. However, they have to stall when they are about to retire and the prior fence is not completed. If an incoming coherence message conflicts with a speculative load, the load is squashed and retried.

Fences prevent access reorderings that, while allowed by the memory consistency model of the platform, can cause SCVs. Recall that SC requires that the memory accesses of a program appear to execute in some global sequence as if the threads where multiplexed on a uniprocessor [17]. An SCV occurs when the memory accesses reorder in a non-SC conforming interleaving. An SCV is typically a harmful bug.

An SCV is caused by two or more overlapping data races where the dependences end up ordered in a cycle [29]. Figure 1a shows the required program pattern for two threads (where each variable x and y is written at least once). Figure 1b shows the required order of the dependences at runtime to cause an SCV (where we assigned reads and writes to the references arbitrarily). Due to access reordering in one of the threads (or in both), *A1* occurs before *B0*, and *B1* occurs before *A0*. The cycle *A1*→*B0*→*B1*→*A0*→*A1* is now created, and this order does not conform to any SC interleaving.

If at least one of the dependences occurs in the opposite direction (e.g., as in Figure 1c), no SCV occurs. To force one

or both dependences to go in the opposite direction, we must place one fence between references *A0* and *A1*, and another between *B0* and *B1* [29] (Figure 1d). It can be seen that, if *A1* is not seen by P_B until *A0* is completed, and *B1* is not seen by P_A until *B0* is completed, no cycle is possible.

This idea extends to any number of threads. For example, Figure 1e shows three threads with accesses that could potentially cause a cycle. To prevent a cycle, we need to prevent the reordering of accesses in each of the threads and, therefore, we need three fences (Figure 1f).

In this paper, we say that two or more fences running on different threads *collide* when their execution overlaps in time. When two or more colliding fences end up preventing a cycle, we say that these fences form a *Fence Group*. Note that a fence group is a dynamic concept.

2.2 Aggressive Techniques to Speed-up Fences

There have been four recent proposals of high-performance fence designs: WeeFence [8], Address-Aware Fences (AAF) [19], Conditional Fences (C-Fences) [20], and Location-Based Memory Fences (l-mfences) [15]. The first two are the most aggressive ones, and directly motivate our work. Hence, we discuss them here. The other two are discussed in detail in the related work section (Section 8).

The ideas in WeeFence [8] and AAF [19] are similar. Post-fence accesses are allowed to *complete* before the fence completes, but only if the resulting reordering is not about to cause an SCV. If it is, these post-fence accesses are stalled until the SCV cannot occur anymore. Since most of the time a fence will not collide with another one to form a fence group, the approach is beneficial. However, to detect if a reordering may cause an SCV, both schemes need *global* state. They need to know if there is any concurrently-executing fence and, if so, the pending pre-fence accesses of such a fence. These are the accesses to watch for to avoid a cycle.

To see the difficulties involved, we describe WeeFence; AAF has similar issues. Figure 2a shows two fences that prevent a cycle. Assume that WeeFence1 is in progress and P1:rd_y tries to complete. P1:rd_y needs to be careful not to cause dependence arrow *B* if there is any chance that an arrow like *A* may already exist. Such pair of arrows would cause a cycle and induce an SCV.

To avoid this case, when a WeeFence starts executing, it collects the addresses of its pending pre-fence accesses (the Pending Set (PS)) and deposits them in a global table called Global Reorder Table (GRT). At the same time, from the GRT, it grabs the PSs of all the currently-executing fences and brings them to a local structure called Remote PS. From then on, every local post-fence access checks its address against the Remote PS. If there is a match, it stalls.

This strategy prevents arrow *B* in Figure 2a. Indeed, if WeeFence2 was executing, it would have left address *y* in the GRT. WeeFence1 would have brought *y* into its Remote PS and P1:rd_y would stall, preventing arrow *B*.

However, this is not all. WeeFence still needs to do more. Figure 2b shows a pattern under TSO where a single fence prevents a cycle. P2 needs no fence because, under TSO,

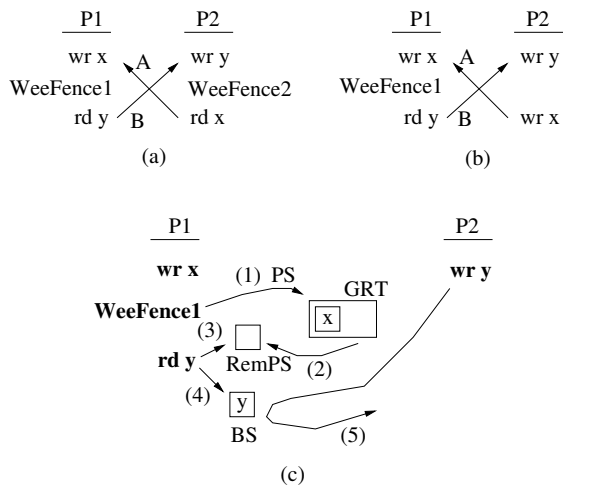

Figure 2. Operation of WeeFence.

there is no write-write reorder. Therefore, WeeFence1 will not find any PS state left in the GRT by any concurrently-executing fence. Therefore, WeeFence1 cannot use the Remote PS to stall P1:rd_y and prevent arrow *B*. However, if we allow arrow *B* to happen, and then arrow *A* happens, we have a cycle. Hence, WeeFence1 has to delay the occurrence of an arrow like *B* with any processor that has not already registered its PS in the GRT when WeeFence1 checks the GRT.

To accomplish this, when a post-WeeFence access does not find a match in its Remote PS, as it executes, it stores its address in a local Bypass Set (BS). The BS is in the cache controller, and all incoming coherence requests will be checked against it. In case of a match, the coherence request is rejected. The BS remains until the local fence completes.

This strategy delays the creation of the arrow *B* in Figure 2b. Assume P1:rd_y completed. As request P2:wr_y reaches P1 and checks the BS, it finds a match and gets bounced. Arrow *B* will not be allowed until WeeFence1 completes, at which point arrow *A* cannot occur.

Figure 2c shows the operation of WeeFence. The execution of WeeFence1 involves collecting its PS (i.e., *x*), storing it in the GRT (1), and bringing the combined PSs of all other active fences into the Remote PS (2). Then, every post-WeeFence1 access compares its address against the Remote PS (3). On a match, the access stalls; else, it executes and puts its address in the BS (4). The access may complete before WeeFence1 completes. Any incoming coherence access is checked against the BS (5). On a match, the incoming transaction bounces.

AAFs [19] work similarly. When an AAF executes, it collects global information on pending accesses in other processors, and brings it into a local Watchlist. Local accesses that hit in the Watchlist stall. The processor has a local Active Buffer like the BS that stalls incoming coherence transactions that hit there. While WeeFence is described for TSO and AAF for RC, both schemes can be adapted to either model.

2.3 Limitations

These two fence designs, while effective, are challenging to implement in a distributed-directory environment. There are

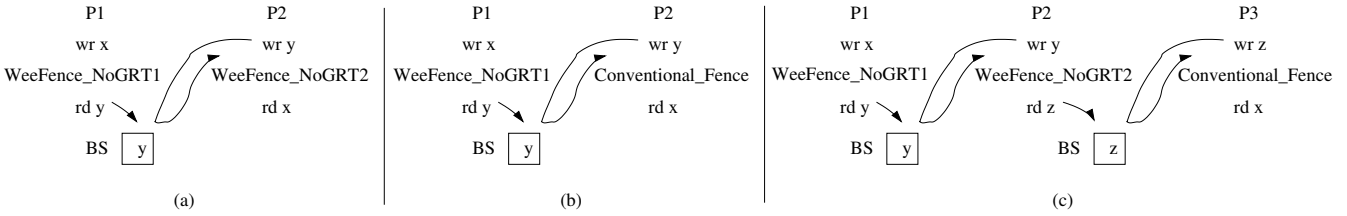

Figure 3. Eliminating global state without suffering from deadlock.

two reasons, which stem from the schemes' reliance on updating and collecting global state. First and foremost, they are subject to having coherence protocol races. In general, a fence needs to collect Remote PS state from different directory modules (since the state is distributed according to physical addresses). Such state needs to be consistent. Unfortunately, multiple processors may be depositing PSes and reading PSes from multiple directory modules with some unknown interleaving. Obtaining a consistent view is hard. We believe that this problem is still unsolved. To avoid this problem, if a WeeFence needs to deposit/access PS to/from more than one directory module, it is turned it into a conventional fence [8] — which lowers performance.

The second reason is that handling the global state adds complexity. It requires: adding new coherence messages to get pending sets (AAF and WeeFence), augmenting protocol messages with address sets (AAF and WeeFence), adding the GRT hardware table in the directory (WeeFence), and collecting addresses into signatures (AAF and WeeFence).

3. Asymmetric Fences

3.1 Main Idea

Our goal is to design a fence architecture that optimizes both performance and hardware implementability. Our contribution is to eliminate the global-state requirements of aggressive fences like WeeFence or AAF, and use the resulting fence in combination with conventional fences. With this, we greatly simplify the implementation and retain the performance. We call this approach *Asymmetric* fences. Next, we describe the ideas in the context of WeeFence.

3.1.1 Minimizing Hardware Cost

The reason why WeeFence needs to save state in the GRT global table is to avoid deadlock when preventing an SCV. If there was no such global state, at the onset of an SCV, the processors would deadlock. This can be seen in Figure 3a, which is WeeFence without GRT or PS. In the figure, P1:rd_y has completed, while P1:wr_x is still pending (and hence address *y* is in P1's BS). Moreover, P2:wr_y is incomplete. As P2:wr_y's transaction is issued into the network, it bounces off P1's BS and keeps retrying. WeeFence_NoGRT2 is then bypassed, and P2:rd_x executes, placing address *x* in P2's BS. The execution of P1:wr_x issues a transaction that bounces off P2's BS and keeps retrying. The system is deadlocked.

In WeeFence, however, P1 deposits its PS (i.e., address *x*) in the GRT while bypassing WeeFence1, and P2 reads the GRT when it finds WeeFence2. Then, P2:rd_x is unable to execute because its address matches what was read from the GRT. Later, P1:wr_x finishes, WeeFence1 completes, P1's BS is cleared, and P2:wr_y can make progress.

Our insight is that, if *at least one of the fences* in the fence group is a conventional fence, there is no need for the GRT or PS state. This is shown in Figure 3b for a 2-fence group, and in Figure 3c for a 3-fence group.

Consider Figure 3b first, where P2 now uses a conventional fence. The execution state is the same as in Figure 3a: P1:rd_y has completed, P1's BS has address *y*, and P2:wr_y is bouncing. However, the conventional fence prevents P2:rd_x from executing *non-speculatively* — i.e., if P2:rd_x executes, it must remain speculative, and a coherence message from P1:wr_x will squash it. As a result, P1:wr_x does not stall. Its completion will complete WeeFence_NoGRT1, clear P1's BS, and enable P2:wr_y to make progress.

Similarly, in Figure 3c, where only P3 uses a conventional fence, there is no deadlock possible. In the worst case, P2:wr_y is stalled by P1's BS and P3:wr_z is stalled by P2's BS. However, P3:rd_x cannot stall P1:wr_x.

In summary, we have transitioned from an N-fence group with all WeeFences, to one where N-1 fences are WeeFences without global state and one is a conventional fence. This simplifies the hardware implementation substantially.

3.1.2 Retaining High Performance

In many cases, a program where individual fence groups have both WeeFence_NoGRTs and conventional fences can deliver as much performance as if all the fence groups only had WeeFences. This is because, in a fence group, there are often one or more threads that execute performance-critical operations, while the other threads do not. Hence, we use WeeFence_NoGRTs in the former threads and conventional fences in the latter. The result is that the overall program performance is the same as if all the threads used WeeFences.

Ladan-Mozes et al. [15] observed that, in a two-fence group, there is sometimes a thread that is more important than the other. In this paper, we consider fence groups with any number of threads.

Two examples where we can combine WeeFence_NoGRT and conventional fences are algorithms in work stealing and software transactional memory (STM). Specifically, in the Cilk runtime system [9], a thread may be dequeuing a task from its task queue Q while a second thread is stealing a task from Q. Both owner and thief use fences to avoid an SCV. Since, typically, the owner dequeues from Q much more frequently than a thief, we use a WeeFence_NoGRT in the owner code and a conventional fence in the thief code.

In STM, there are fences when threads read a variable, write a variable, and commit a transaction. In the STM scheme that we use later, when a thread that reads a variable conflicts with another that writes the same variable, their fences prevent an SCV. Since reads are more frequent and time-critical than writes, we use a WeeFence_NoGRT in the read code and a conventional fence in the write code.

3.2 Strong Fence and Weak Fence

We define an *Asymmetric* fence group as one that is composed of one or more *Strong Fences* (*sF*s) and one or more *Weak Fences* (*wF*s). An *sF* is a conventional fence. It allows post-fence reads to execute speculatively, but not to complete, before the fence completes. On a conflict with an incoming coherence message, a speculative read is squashed.

A *wF* is a WeeFence with no GRT or PS, augmented with a few small additions that we will describe. It allows the same post-fence accesses as WeeFence to execute, retire, and complete before the fence completes. The addresses referenced by post-fence accesses are put in the BS. When one such access cannot be squashed anymore, the BS rejects incoming requests that conflict with its address. In TSO, which is the focus of this paper, the following holds: (i) the accesses in the BS are *post-fence reads*; (ii) these reads cannot be squashed anymore after they retire; and (iii) the rejected incoming requests are *write* transactions that attempt to *invalidate* the line. In other consistency models, other conditions apply.

Recall from WeeFence [8] that the BS is stored in a hardware list in the cache controller, and that it can include a front-end Bloom-filter to reduce the number of comparisons.

BS addresses and coherence transaction addresses are compared at line granularity. This is because the coherence protocol, which detects the dependences, uses line addresses. Figure 4a shows why using finer-grain addresses (e.g., word-level) would be incorrect. The example is like Figure 3b, except that P2 writes to word *y'* before writing to *y*, where words *y'* and *y* share the same line. If the comparison between the BS and the transaction (1) induced by P2:wr_y' was done at word granularity, there would be no match. Hence, the line would be brought to P2 and, later, P2:wr_y would complete execution locally, potentially causing an SCV.

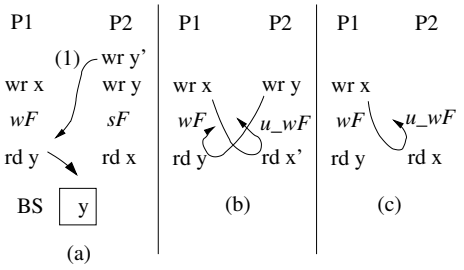

Figure 4. Examples using Asymmetric fences. In the figure, u_wF means *unrelated* Weak fence.

We expect Asymmetric fences to be used in codes that require high performance — possibly in libraries such as those for work-stealing scheduling, STM, or synchronization. These codes are typically programmed by expert programmers. It is reasonable for these programmers, for example, to place a *wF* in the code of the owner thread in a work-stealing runtime, and an *sF* in the code of the thief thread.

However, it is unreasonable for these programmers to know or worry about false sharing. Consequently, when two or more unrelated *wF*s whose pre- and post-fence accesses could form a cycle due to false sharing end up executing concurrently, the hardware has to work seamlessly. With the microarchitecture that we propose in Section 3.3, the programmer is unaware of any false-sharing related effects.

In the following, we present a taxonomy of Asymmetric fence groups, and describe the implementation of the *wF*s for the different design points. We assume the TSO model.

3.3 Taxonomy of Asymmetric Fence Groups in TSO

We design the *wF* slightly differently depending on our assumptions on what Asymmetric fence groups are possible. We propose three *wF* designs: *WS+*, *SW+*, and *W+*. First, *WS+* is the preferred design if we can assume that all Asymmetric fence groups will include at most one *wF* — i.e., the rest of the fences in the fence group will be *sF*s. Next, *SW+* is a design that works for all Asymmetric fence groups. It relies on the presence of at least one *sF* in the fence group to avoid deadlocks. Finally, *W+* is a design that works for all Asymmetric fence groups and even when *all* the fences in the group are *wF* — which, strictly speaking, is not an Asymmetric fence group anymore.

For comparison, we also consider two known environments: WeeFence and the conventional case where all fence groups only contain *sF*s (*S+*). Table 1 lists all the environments. The *S+* design has the lowest hardware complexity and the lowest performance. At the other extreme, the design with all-WeeFence fence groups (*Wee*) has the highest complexity because it uses global state. Next, we present our proposed designs.

3.3.1 At Most One Weak Fence in the Group (*WS+*)

If we can guarantee that any Asymmetric fence group will have at most one *wF*, the design of the *wF* requires relatively minor changes over a WeeFence without GRT or PS. The reason is that, in these groups, the accesses preceding the *wF* *never* need to bounce-off from another processor's BS to prevent SCVs. This is because the other processors participating in the fence group execute *sF*s, which have no BS.

Therefore, if, at runtime, an access preceding a *wF* bounces, it is due to interference with an *unrelated wF* that happens to be executing concurrently. Such interference cannot create an SCV and, hence, can be handled as such.

Specifically, the bouncing of a pre-*wF* write can be due to two cases. The first one is a cycle with another *wF* due to false sharing. This is shown in Figure 4b, where *x* and *x'* are two words from the same line. P1:wr_x bounces off P2's BS because P2:rd_x' has completed. Recall that the comparison between BS addresses and coherence transaction addresses is done at line granularity. In this case, the threads could deadlock and the bouncing continue indefinitely. The second case is a short-lived bouncing due to a true-sharing (or

wF Design Point		Hardware Support Required
Name	Corresponding Fence Group	
S+	Fence groups with only *sF*s	None (conventional fence)
WS+	Asymmetric groups with at most one *wF*	BS, Order bit, and Order operation
SW+	Any Asymmetric group	BS, Order bit, fine-grain info, and Conditional Order operation
W+	Any Asymmetric group and *wF*-only groups	BS, checkpoint, detect bouncing & being bounced, timeout, and recovery
Wee	WeeFence [8]	BS and global state (GRT and PS)

Table 1. *wF* designs with a taxonomy of Asymmetric fence groups under TSO.

false-sharing) dependence that does not cause a cycle. This is shown in Figure 4c. In all of these cases, no SCV is possible.

To handle these cases, we cannot simply transform the bouncing transaction into a plain coherence one — e.g., in Figure 4c, by letting P1:wr_x invalidate the line from P2's cache and bringing it to P1's cache. The reason is that P2's BS still has to see all the future coherence transactions directed to *x*. Invalidating the line from P2's cache would prevent that.

Our goal is to ensure both that (i) P1 makes progress (i.e., P1:wr_x completes, getting ordered after P2:rd_x), and (ii) P2's BS keeps its ability to monitor all the future coherence transactions directed to *x*. We accomplish this with the *Order* operation, which *orders* P1:wr_x after P2:rd_x, but allows P2 to retain its monitoring ability on *x*.

Specifically, we augment request messages with a bit called Order (*O*). Typically, *O* is zero. Assume that a request issued by P1 reaches the BS of P2 and there is a match (at line granularity). The transaction bounces and keeps retrying. If P1 then executes a *wF*, we know that the bouncing is unneeded, and the hardware sets the *O* bit of all of P1's currently-bouncing requests. Each of these write requests becomes an *Order* request in its next retry.

An Order request carries its update in the message. When the request reaches the directory, the latter sends an invalidation to all the sharers. The sharers invalidate the line but, in their response, tell the directory if they still have the line's address in their BS. Those that do are kept as sharers in the directory. This ensures that they will see future coherence accesses to the line. Also, if the line was dirty in a cache, it is written back to memory. The directory returns the line to the requester (or just an ack if the requester was a sharer), and merges the requester's update into memory. On reception of the response from the directory, the requester merges its update into the line and keeps the line in Shared cached state.

Overall, going back to Figure 4c, we have completed P1:wr_x and kept P2 as a sharer of the line, allowing it to see future writes to *x*. For as long as P2 has the address of *x*'s line in its BS, P2 will see any external write transaction to *x*. If the transaction's *O* is clear, it is bounced; if it is set, P2 asks the directory to keep P2 as sharer. In one of these external writes, P2 will not have the address of *x*'s line in its BS anymore, and not tell the directory to keep P2 as sharer.

Note that if P1's bouncing writes are followed by an *sF*, no special action is taken; *O* is kept zero and bouncing continues. Also, recall that the programmer guarantees that the execution will not find any other type of Asymmetric fence group. If this is incorrect, an SCV may silently occur.

Table 1 shows that the *WS+ wF* is a WeeFence without GRT or PS, plus the Order bit and the Order operation.

3.3.2 Any Asymmetric Fence Group (*SW+*)

To handle any Asymmetric fence group, we require a more advanced *wF* design that we call *SW+* (named after the most challenging case of only one *sF* and many *wF*s in the group). The reason why we cannot reuse the *WS+* design is because some pre-*wF* accesses may now need to bounce for correctness. This is the case for the fence in P2 in Figure 3c. P2's pre-fence access needs to bounce until another processor in the group (P1 in the example) completes its fence, clears its BS, and enables P2's progress. Progress is guaranteed thanks to the presence of an *sF* in the group. If, instead of bouncing, we set the Order bit in P2's pre-fence request, we force an order that can cause a cycle and, hence, an SCV.

At the same time, as described in Section 3.3.1, pre-*wF* accesses may interfere with unrelated, concurrent *wF*s and experience unnecessary bouncing. The bouncing can be indefinite when there is a dependence cycle due to false sharing (Figure 4b), or short-lived when there is a true- or false-sharing dependence that does not cause a cycle (Figure 4c).

We solve this problem by continuing to bounce when there is a true dependence between threads, and by triggering an Order operation when the bouncing is due to false sharing. We call this idea *Conditional Order*. Note that some true-sharing induced bouncing may be unnecessary. However, such bouncing is short-lived and eventually stops.

The required hardware support is two-fold. First, the BS now keeps fine-grain addresses — i.e., those of the words (or bytes) accessed. Second, requests contain the *O* bit and a bitmask with as many bits as words (or bytes) in a line.

By default, the BS is checked against external coherence transactions at line granularity. Assume that a write issued by P1 reaches P2's BS and there is a match at line granularity. The request bounces and continues retrying as usual. However, if P1 then executes a *wF*, P1's hardware changes each pre-*wF* bouncing request as follows: (i) it sets the *O* bit and (ii) it sets the bits in the bitmask for the words (or bytes) in the line that are being requested. Note that it is possible that P1 requests multiple words of the same line; the requests are combined into a single one. These changes transform the bouncing request into a Conditional Order (CO) request.

A CO request starts-off as an Order request: all the sharers are forced to invalidate their cached copies of the line. However, the sharers that have the line's address in their BS tell the directory if the *match is due to true or false sharing*. Both types of processors are kept as sharers in the directory. How-

ever, the directory proceeds differently depending on whether there are any true-sharers at all among them.

If there are not, the CO transaction completes as an Order transaction in *WS+*. Otherwise, the CO transaction fails and bounces back to the requester, and the hardware retries it again as a CO request. Moreover, the directory discards the requester's update. Overall, the failed CO request had no effect except invalidating the caches; the processors with matching line-addresses in the BS are still sharers.

Eventually, the BSes with true-sharing addresses will clear: in the case of a normal fence group, thanks to having an *sF* in the group; in the case of a short-lived interference with an unrelated *wF*, when the latter completes. After that, when the sharers receive invalidations, they will inform the directory that all BSes match only due to false sharing. Then, the transaction will complete as an Order transaction.

Again, if P1's bouncing writes are followed by an *sF*, no special action is taken and bouncing continues. We could stop bouncing if it was false sharing, but such an optimization is unnecessary, given that *sF*s are used by non-critical threads.

Table 1 shows that the *wF* in *SW+* is a WeeFence without GRT or PS, plus the Order bit, the fine-grain address information in the BS and in requests, and the CO operation.

3.3.3 All Fences in the Group Can Be Weak (*W+*)

The *W+* design supports all Asymmetric fence groups and all fence groups where all the fences are *wF*s. As discussed in Section 3.1.1, a *wF*-only group, where *wF*s are implemented as WeeFences without GRT or PS, ends up deadlocking as it prevents an SCV. In *W+*, we allow the hardware to deadlock, trigger a time out, rollback the state to before the *wF*s, and retry execution while avoiding the deadlock again.

W+ does not distinguish between genuine fence groups, and cycles due to false sharing. In all cases, when multiple colliding *wF*s end up trying to avoid a cycle, the threads will deadlock. The recovery process is the same.

In TSO, recovery is not too costly. Since the post-*wF* accesses that can complete before the *wF* completes are necessarily loads, we can recover by mostly reusing mechanisms already present in current processors. However, recovery in models such as RC would be costly.

The hardware implementation is as follows. *wF*s are largely WeeFences without GRT or PS. There is no fine-grain address information — i.e., both BS and request transactions use line-level addresses. As usual, pre-*wF* writes bounce if they hit in another processor's BS, and post-*wF* reads can complete before the *wF*. However, a difference is that, when a *wF* reaches the head of the ROB, the hardware takes a register checkpoint, in case a rollback is later needed. Note that there may be many pending pre-*wF* writes.

After the checkpoint creation, as soon as the hardware detects that (1) at least one pre-fence write is being bounced and (2) the BS bounces external requests, it starts a timeout. When the timeout goes off, the hardware assumes there is a deadlock. Hence, it restores the checkpoint and clears the BS. This brings the processor to right at the *wF*. At this point, the processor waits until its write buffer is drained, which completes all the pre-*wF* accesses, and then resumes execution. The same deadlock is not possible anymore.

As shown in Table 1, the *wF* in *W+* needs support for checkpointing, detecting when a processor's requests are being bounced and the processor bounces requests, timing out, and performing a rollback recovery — all in hardware.

4. Examples of Asymmetric Fence Uses

4.1 Runtime Schedulers with Work Stealing

Cilk, TBB, and other runtime schedulers use work stealing. In work stealing, each thread owns a task queue. A thread removes (*take()*) tasks from the tail to execute. It may also append new tasks to the tail. When the queue becomes empty, a thread tries to steal a task from the head of the queue of another thread. Hence, *take()* and *steal()* may conflict with each other. To coordinate them without expensive synchronization, the Cilk THE algorithm [9] adopts a Dekker-like protocol, as shown in Figure 5a.

take()	*steal()*
Tail = t	Head = h
fence	fence
h = Head	t = Tail

(a)

read(M,tid)	*write(M,tid)*
Lock(M).readers[tid] = 1	Lock(M).writer = tid
fence	fence
w = Lock(M).writer	r = Lock(M).readers

(b)

Figure 5. Examples from work stealing (a) and STM (b).

In *take()*, the worker first decrements the tail pointer, then checks the head to see if anyone is trying to steal the same task. If so, it will fall into a lock-based path to compete with the thief; if not, it will take the task. In *steal()*, the thief first increments the head pointer, then checks the tail to see if the owner is trying to take the task. If not, it steals the task. The protocol works if: (1) the thief observes the worker decrementing the tail before it observes the worker performing the check, and (2) the worker observes the thief incrementing the head before it observes the thief performing the check. Otherwise, an SCV can occur and a task can be executed multiple times.

To ensure the two requirements, the protocol needs two fences like in Figure 5a. Such fences are typically unnecessary because very little stealing occurs — in our workloads we see less than 0.5% of the total tasks being stolen. However, the fences must be there for correctness. Unfortunately, they cause an average of ≈15% execution time overhead.

These fences can form two-fence groups. We can use Asymmetric fences to optimize them. For example, since the worker executes much more frequently than the thief, we can use a *wF* in the former and an *sF* in the latter.

4.2 Software Transactional Memory

To enable optimistic concurrency between threads that might make conflicting accesses to shared-memory locations, STM programs enclose accesses inside Read and Write Barriers. These are software routines that, in addition to performing the requested read or write, also update the STM metadata to ensure proper serialization of the transactions. Typically,

these metadata accesses use ad-hoc synchronization mechanisms that rely on fences.

We use the open-source Rochester Software Transactional Memory (RSTM) library [1], and consider its implementation of the TLRW algorithm [6]. TLRW is an eager-locking, eager-versioning algorithm based on read/write locks. There is one lock per shared-memory location. Each lock object has two parts: (1) an array of per-thread "reader" flags, and (2) a "writer" field. Hence, there can be multiple readers or a single writer for a memory location.

These locks are used to detect conflicts when performing transactional accesses. In Figure 5b, *M* is the memory location being accessed transactionally, *Lock(M)* is its lock metadata, and *tid* is the ID of the thread performing the access. A reading transaction writes its "reader" flag and then checks the "writer" field to see if there is any concurrent writer. A writing transaction writes to the "writer" field and then reads all the "reader" flags to determine if there are any concurrent readers. To be correct, these accesses have to be made visible to other threads in program order. Hence, fences are used.

The fences in a read and a write operation can form two-fence groups. Typically, reads are considerably more frequent than writes (3.5x in our workloads). Thus, we can use a *wF* in *read()* and an *sF* in *write()*.

4.3 Bakery Algorithm

Lamport's Bakery algorithm [16] is a lock-free mutual-exclusion algorithm of an arbitrary number of threads. It simulates a baker's shop where each customer grabs an increasing number and waits for his turn to be serviced. The algorithm uses two shared arrays (*E* and *N*), each with as many entries as threads. *E[i]* denotes whether thread *i* is trying to grab a number, while *N[i]* is the number currently held by *i*. A thread grabs a number, waits for its turn, goes to execute some critical section, and then repeats.

Figure 6a shows a code snippet with a fence. The code is executed by all threads. First, a thread writes its own *E* entry (*E[ownpid]*) and then, in a loop, goes on to read the other threads' entries (*E[pid]*). The execution can induce fence groups with any combination of thread count and pid — e.g., Figures 6b and 6c show a group with threads *T0* and *T2*, and one with threads *T4*, *T1*, and *T3*, respectively.

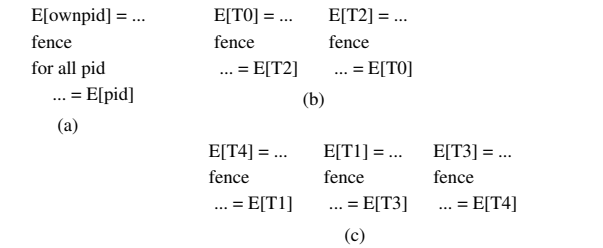

Figure 6. Asymmetric fences in the Bakery algorithm.

If we want to give priority to one thread, Bakery can use *WS+*. For example, if we want to give priority to *T0*, then, we use a *wF* in its code, while we use an *sF* in the other threads' code. *T0* will execute faster than the others, and we will observe *WS+* fence groups every time that *T0* participates in one of them. On the other hand, if we want all threads to run equally fast, we can use *W+*.

4.4 Other Algorithms and Domains

There are other algorithms and domains where Asymmetric fences can be used. One example is distributed lock-free lists, queues, or other structures. Another is many aspects of STM libraries. Since such libraries come in many flavors (e.g., eager or lazy, or optimized for performance or for readability) and are written by experts, they are a promising area. Other examples are environments that use biased locking such as Java Monitors [5, 14] and garbage collectors in a Java Virtual Machine (JVM). Such locking may be translated into Asymmetric fences. Finally, another example is double-checked locking [28] under relaxed memory models.

5. Discussion

5.1 Hardware Implementation Issues

The *wF* designs that we use in this paper are largely WeeFence for TSO without GRT or PS [8], plus relatively small modifications. Note that, under TSO, a BS entry only rejects incoming matching coherence transactions that attempt to *invalidate* the local line. Incoming read transactions are always serviced, even if they downgrade a local cached line from Dirty to Shared. Such downgrade does not hurt the BS ability to intercept future external writes to the line. We did not explicitly made this point in [8].

However, our *wF* designs differ from WeeFence without GRT or PS in one aspect: the handling of cache evictions of Dirty lines whose address happens to be in the BS. In [8], such evictions required storing the line's address in the GRT. Now, there is no GRT. Hence, we use the support described in Section 3.3.1, where a cache can get invalidated but, if it has the line's address in its BS, it requests the directory to keep the node as sharer. This support was described in the context of Order transactions.

Hence, in all of our designs (*WS+*, *SW+*, and *W+*), when a Dirty line is displaced and its address is in the BS, we do the following. As the line is written back, the cache requests the directory to keep it as sharer, so that it can see future writes to it (and can potentially bounce them). Note that evictions of clean lines are not a problem. Since they are silent, the directory still considers the displacing cache a sharer.

Overall, our three designs significantly simplify the hardware over WeeFence, by eliminating global hardware and state. Each design has slightly different hardware requirements, as shown in Table 1.

5.2 Implementing Asymmetric Fences in RC

Supporting Asymmetric fences in RC requires redesigning the *wF* implementations. This is because, in RC, when the *wF* executes, there may be incomplete pre-fence writes and reads. Moreover, before the *wF* completes, the post-fence accesses in the BS may include reads executed and writes potentially merged with the memory system.

As an example, the *W+* design under RC will need to recover from completed post-*wF* reads and writes. Hence, after the *W+* hardware creates a register checkpoint at the *wF*, it needs to buffer the post-*wF* writes that complete early. One approach is to place such writes into a speculative buffer or cache, while the regular cache issues exclusive prefetches for the lines. If rollback is required, the checkpoint is restored and the state in this speculative buffer or cache is discarded. We leave the design of *wF*s under RC for future work.

5.3 Programming Challenges

While this paper focuses on hardware issues, we note that *wF*s introduce programming challenges. In this section, we discuss two of them. Addressing them is beyond our scope.

One challenge is that all of the *wF*s and *sF*s that participate in a given fence group have to be contained in a code region or library that the programmer can realistically understand. If the programmer is unaware of the full extent of a possible fence group, the code may operate incorrectly. Consequently, fence groups in code that crosses software module and abstraction boundaries are unlikely to be good candidates for Asymmetric fences. Future work involves developing tools that help programmers isolate fence groups and avoid incorrect code. It also involves understanding what classes of algorithms are suitable for Asymmetric fences.

A second issue occurs with code that has an SCV and still functions as intended. For example, Figure 7a shows two threads that first release a lock and then acquire another one. Assume that a release is implemented as a store, an acquire as a test-and-test&set (using an exchange instruction), and that value 0 is free and 1 is taken. We have the code in Figure 7b. The first two accesses of this code are shown in Figure 7c. If there is an SCV, the code still works as intended.

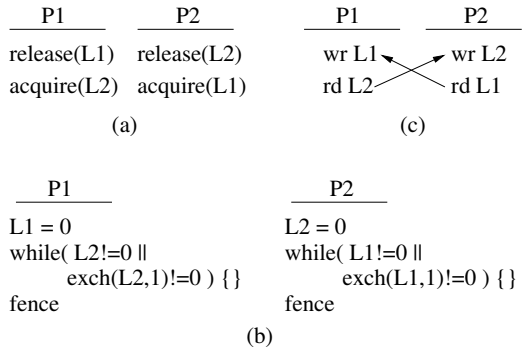

Figure 7. Code with an existing SCV.

Our Asymmetric fence designs assume that the input code does not have SCVs to start with. Consequently, they may not work with the code in Figure 7c. For example, assume that, in between P1:wr_L1 and P1:rd_L2, there is code with a *wF* and, in between P2:wr_L2 and P2:rd_L1 there is code with an unrelated *wF*. If these *wF*s are implemented as *SW+*, the system may deadlock as both *wF*s attempt Conditional Order operations. On the other hand, if they are implemented as either *WS+* or *W+*, the code executes correctly. Future work involves examining the interaction of *wF* implementations with codes that contain SCVs.

Overall, we expect Asymmetric fences to be used mostly by expert programmers, as they develop performance-sensitive codes — e.g., synchronization, TM, or task scheduling libraries.

6. Evaluation Setup

To evaluate Asymmetric fences, we use detailed cycle-level execution-driven simulations. We model a multicore with 8 cores connected in a mesh network with a directory-based MESI coherence protocol under TSO. Each core has a private L1 cache, a bank of a shared L2 cache, a portion of the directory and, for WeeFence, a module of the distributed GRT. For some experiments, we change the number of cores from 4 to 32. Table 2 shows the architecture parameters. For WeeFence, we use the default parameters in [8].

Architecture	Multicore with 4-32 cores (default is 8)
Core	Out of order, 4-issue wide, 2.0 GHz
ROB; write buffer	140 entries; 64 entries
L1 cache	Private 32KB WB, 4-way, 2-cycle RT, 32B lines
L2 cache	Shared with per-core 128KB WB banks A bank: 8-way, 11-cycle RT (local), 32B lines
Bypass Set (BS)	Up to 32 entries per core, 4B per entry
Cache coherence	MESI under TSO, full-mapped NUMA directory
On-chip network	2D-mesh, 5 cycles/hop, 256-bit links
Off-chip memory	Connected to one network port, 200-cycle RT

Table 2. Architecture modeled. RT means round trip.

We tune our simulator so that a conventional fence has approximately the same overhead as indicated in [8] for a desktop with a 4-core Intel Xeon E5530 processor.

For the evaluation, we use three workload groups. They are listed in Table 3. The first one is a set of Cilk applications (*CilkApps*) that use the THE work-stealing algorithm [9]. As indicated in Section 4.1, all fence groups are formed by 2 fences, one in the worker code and one in the code for the thief. For both *SW+* and *WS+*, we use a *wF* in the worker code and an *sF* in the thief code.

Workload Group	Applications
Cilk Apps. (*CilkApps*)	bucket, cholesky, cilksort, fft, fib, heat, knapsack, lu, matmul, plu
STM Micro- benchs. (*uSTM*)	Counter, DList, Forest, Hash, List, MCAS, ReadNWrite1, ReadWriteN, Tree, TreeOverwrite
STAMP Apps.	genome,intruder,kmeans,labyrinth,ssca2,vacation

Table 3. Applications used in the evaluation.

The second workload is a set of STM microbenchmarks (*uSTM*). They are obtained from the Rochester Software Transactional Memory (RSTM) library [1] and use the TLRW algorithm discussed in Section 4.2. Each microbenchmark consists of a concurrent data structure and transactions that look-up, insert, or delete data in the structure. 50% of the transactions are lookups, and the rest are equally divided between insertions and deletions. As per Section 4.2, fence groups are formed by two fences, one in the read operation and one in the write operation. For both *SW+* and *WS+*, we use a *wF* in the read code and an *sF* in the write code.

The third workload has applications from the STAMP suite [23] distributed with RSTM. The fence groups and the assignment of *wF* and *sF* are the same as in *uSTM*.

In both *CilkApps* and *STAMP*, we report performance as execution time. For *uSTM*, since there is no standard input set, we run each microbenchmark for a certain fixed time and measure the number of transactions committed. We report performance as throughput.

We find that the performance of our workloads under *SW+* and *WS+* is practically the same. This is unsurprising, given that our fence groups have two fences. Hence, to simplify the evaluation, we do not show data for *SW+*.

Recall that *sF*s allow *speculative* execution of post-fence reads. Also, when a WeeFence cannot confine its PS and BS into a single directory module, it turns into an *sF* [8].

7. Evaluation

7.1 Performance Comparison

Figure 8 compares the execution time of *CilkApps* for different types of fences. For each application, we show, from left to right, bars for *S+*, *WS+*, *W+* and *Wee* fences, all normalized to *S+*. The time is broken down according to whether the processor is retiring instructions (*Busy*), is stalled for fences (*Fence Stall*) or is stalled for other reasons such as memory or pipeline hazards (*Other Stall*). The rightmost set of four bars shows the average of all applications.

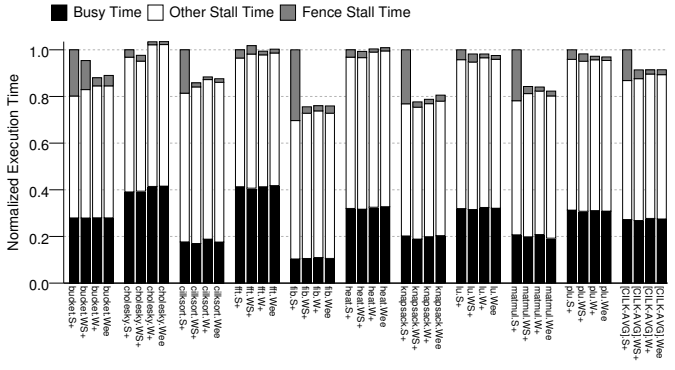

Figure 8. Execution time of *CilkApps*.

Looking at the average bars, we see that, with conventional fences (*S+*), *CilkApps* spend 13% of their time stalled in fences. *WS+*, *W+* and *Wee* eliminate most of such stall. With these designs, the remaining fence stall time amounts to an average of 2-4% of the application time. The result is that, with either of these three designs, the overall execution time of *CilkApps* is reduced by an average of 9%.

The overall average performance impact is necessarily limited by the average fraction of original time spent on fence stall. However, we see that there are applications with 20-30% of the time spent on fence stall and, in those cases, *WS+* and *W+* eliminate most of it. As fence stall decreases, other stall sometimes increases — e.g., memory operations that bypass fences then induce memory contention.

WS+, *W+* and *Wee* perform similarly because, in work-stealing, most of the executed fences are *wF*. Moreover, there are very few recoveries in *W+*. Overall, *WS+* are *W+* are equally attractive and much more cost-effective than *Wee*.

We now consider the *uSTM* workload. As shown in Figure 9, we measure performance as transactional throughput — i.e., the number of transactions committed per second. Therefore, higher is better. For each microbenchmark, we show bars for *S+*, *WS+*, *W+* and *Wee*, all normalized to *S+*. The rightmost set of bars is the average.

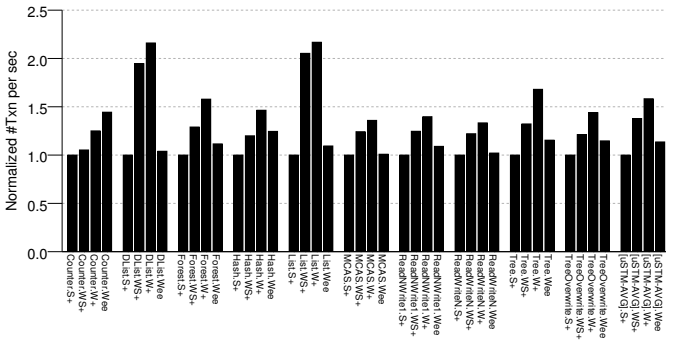

Figure 9. Transactional throughput of *uSTM*.

As shown in the figure, *WS+*, *W+* and *Wee* all outperform *S+*. This is because, by reducing the fence stall time, these designs are able to speed-up the execution. On average, *WS+*, *W+* and *Wee* increase the transactional throughput by 38%, 58%, and 14%, respectively, over *S+*. We see that *W+* and *WS+* are much more cost-effective than *Wee*.

To understand these results better, Figure 10 shows the per-transaction breakdown of processor cycles. This figure breaks down the bars into the usual categories. Compared to *CilkApps* in Figure 8, these microkernels spend a much higher fraction of their time in fence stall. On average, in *S+*, *uSTM* spend 54% of their time stalled in fences.

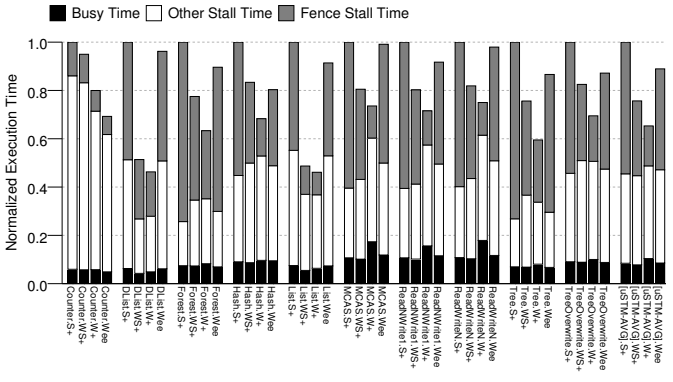

Figure 10. Per-transaction breakdown of processor cycles.

The figure shows that Asymmetric fences are very effective. *WS+* and *W+* eliminate half and two thirds of the average fence stall time, respectively. As a result, the average transaction takes 24% and 35% fewer cycles in *WS+* and *W+*, respectively, than in *S+*. *W+* reduces more fence stall time than *WS+*. However, in part because of its deadlock recoveries, it has a higher busy time than *WS+*. Interestingly, *Wee* reduces the fence stall time little. The reason is that *Wee* ends-up turning many of its fences into *sF*s. *Wee* only manages to reduce the average transaction time by 11%.

Workload	*S+*	*WS+*						*W+*			*Wee*		
	#*sF*s /1000i	#*sF*s /1000i	#*wF*s /1000i	#lines /BS	#wr bounc /*wF*	#retries /wr	%traffic incr.	#*wF*s /1000i	#recov. /*wF*	%traffic incr.	#*sF*s /1000i	#*wF*s /1000i	#lines /BS
CilkApps	1.1	0.3	0.8	4.7	0.1	1.3	3.6	1.1	0.0	1.1	0.0	1.1	4.6
uSTM	5.7	1.8	4.5	2.6	0.3	0.6	2.3	6.5	0.2	1.3	3.1	2.9	2.4
STAMP	1.3	0.6	0.7	2.5	0.0	0.6	0.6	1.7	0.0	0.9	0.6	1.1	2.4

Table 4. Characterization of Asymmetric fences.

Finally, Figure 11 compares the execution time of *STAMP* applications for different fence types. The bars are broken down as usual. In the figure, we see a lot of variation. This is because each application's potential depends on the amount and type of transactional work that it does.

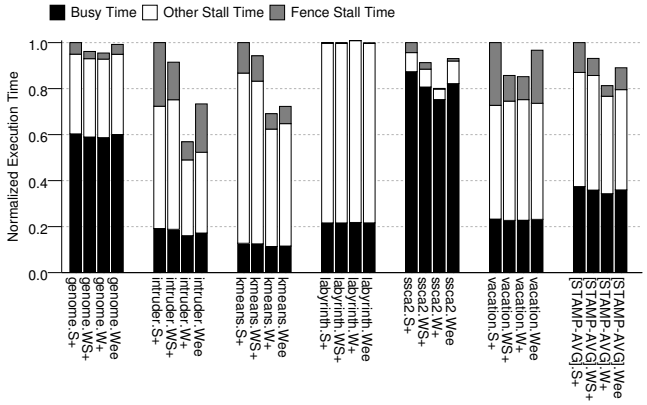

Figure 11. Execution time of STAMP.

For example, *intruder* includes many write operations and, hence, *W+* decreases the fence stall time more than *WS+*. Its changes in fence stall time also affect the other stall time. *Labyrinth* has very few transactions in the first place, and hence cannot get noticeable improvements. *Genome* sees moderate improvements because most of its stall time is due to reasons other than fences. On average, with conventional fences, these applications spend 13% of their time stalled in fences. *WS+*, *W+* and *Wee* reduce the average execution time by 7%, 19%, and 11%, respectively.

Based on the many workloads analyzed, we conclude that, under TSO, *W+* is faster than *WS+*. Across all the workload sets, *W+* and *WS+* reduce the execution time over *S+* by an average of 21% and 13%, respectively. The selection of which scheme is more cost-effective depends on implementation issues. As shown in Table 1, *WS+* requires the Order bit and operation, while *W+* requires detecting a potential cycle, triggering a timeout, and supporting checkpointed recovery. In any case, both schemes are much more cost-effective than *Wee*, which only reduces the execution time by an average of 10% and is more complex.

7.2 Performance Characterization

Table 4 characterizes the *S+*, *WS+*, *W+*, and *Wee* designs for 8 processors. Column 2 shows the average number of *sF*s per 1,000 dynamic instructions in *S+*. For *CilkApps* and *STAMP*, the number is around 1, while for *uSTM* this number is 5.7. Such higher fence frequency is why *WS+* and *W+* get better speedups in *uSTM*.

The next few columns correspond to *WS+*. Columns 3-4 show the average number of *sF*s and *wF*s per 1,000 instructions. The sum of Columns 3 and 4 is not equal to Column 2 for *uSTM* because the *uSTM* experiments measure throughput and execute slightly different code every time. In *STAMP*, *sF*s are about as frequent as *wF*s. Hence, there is a bigger performance gap between *W+* and *WS+* in *STAMP*.

Column 5 shows the average number of line addresses in the BS of a *wF*. We see that this value is 3-5 for the workloads. It can be shown that it corresponds to 12-24 different word addresses. Columns 6-7 consider an average *wF* and show the average number of writes that bounce off it and, for each of these writes, the average number of retries until it can commit. In all cases, the two numbers are low. Hence, the stalls caused by bouncing are largely hidden by the write buffer and do not cause pipeline stall. Column 8 shows the increase in bytes transferred in the network due to write retries. We can see that the increase is negligible.

The next three columns correspond to *W+*. Column 9 shows the average number of *wF*s per 1,000 instructions. Recall that *W+* does not have any *sF*s. Column 10 shows the average number of recoveries per *wF*. This number is only noticeable for *uSTM*, which causes a slightly higher busy time for *W+* in Figure 10. Column 11 shows the increase in network traffic due to *W+*, which is again negligible.

Finally, we show data for *Wee*. Columns 12-13 show the average number of *sF*s and *wF*s per 1,000 instructions. Recall that *Wee* only has the equivalent of *wF*s. However, when a fence's PS and BS cannot be confined into a single directory module, the fence becomes an *sF*. We see that, for *CilkApps*, fences remain *wF*s. However, for *uSTM*, about half of the fences turn into *sF*s. For *STAMP*, about one third do. This effect explains why *Wee* has a higher fence stall time than *WS+* and *W+* in Figures 10 and 11. Column 14 shows the number of line addresses in the BS. These values are similar to those for *WS+* (Column 5). They are higher than in [8] because, in that paper, we used Private Access Filtering. Finally, it can be shown that the number of writes that get bounced per *wF*, and number of retries until a bouncing write can commit are very small, as in *WS+* (Columns 6-7).

7.3 Scalability Analysis

Finally, we assess the scalability of Asymmetric fences' effectiveness to reduce fence stall time. For a given design (say, *WS+*), we compare its fence stall time to that of *S+*, and compute the ratio (stall*WS+*/stall*S+*). This ratio is shown in Figure 12 for different numbers of cores. The figure organizes the data per workload and fence design. For each case, it shows bars for 4, 8, 16, and 32-core runs.

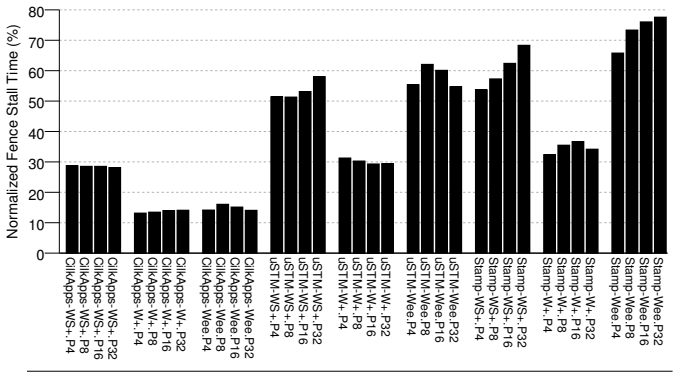

Figure 12. Scalability of Asymmetric fence stall time.

For a given workload and fence design, as we go from 4 to 32 cores, the bars remain flat or increase only modestly. For example, for *CilkApps* with *WS+*, the bars remain at 28%. While the total fence stall time for *CilkApps* with *S+* may change with the core count, *WS+* manages to reduce it always to about 28% of it. This means that *WS+* is scalable. Its effectiveness is not reduced with higher core counts.

Overall, while the various designs have different impacts on different loads, they all keep their effectiveness across different core counts. Hence, Asymmetric fences are scalable.

8. Related Work

Location-based memory fences (l-mfences) [15] is a design that speeds-up the execution of a fence in a thread when no other thread accesses the location protected by the fence often. It is implemented with instructions like Load-Linked (LL) and Store-Conditional (SC). The operation takes as arguments the address of a write that precedes the fence and the value it wants to write. If the memory line accessed by the operation is in the cache in Exclusive state, the operation only involves a cached load and a store. However, if a second thread has accessed the location in the meantime, when the first thread tries to access it again, its SC fails, and it has to perform a conventional fence.

*wF*s and l-mfences have four main differences. First, *wF*s allow *reordering* of accesses across fences: post-fence accesses can *complete* before pre-fence accesses complete. In l-mfences, the SC has to complete before post-fence accesses can complete. Second, *wF*s are more general: an l-mfence takes as argument a write, while a *wF* protects many writes. In addition, the l-mfence only works well if a fence protects the *same* address across invocations. Third, with l-mfences, every time that another thread accesses the location, the line's coherence state changes, and a future l-mfence will fail. A *wF* works well no matter how many times another *sF* executes. Finally, the l-mfence design focuses on two conflicting threads only, while our *wF*s work for any fence group size.

The idea of Conditional Fences (C-Fences) [20] is for the compiler or user to statically classify fences into groups called Associates. These are fences that may appear in a fence group at runtime. At runtime, when a fence executes, the hardware checks a centralized table to see if any other associate fence is executing. If so, the fence stalls until its associate completes. This scheme requires global hardware, and such hardware is centralized. *wF*s eliminate global hardware and any centralization points. It is unclear how the difficulty of grouping fences into associates compares to that of choosing *wF*s and *sF*s. However, *wF*s are compatible with conventional fences in other code modules, but C-Fences are not.

A related approach is post-retirement speculation (e.g., [3, 4, 11, 26]) — a technique that tries to reduce stalls due to access reorders disallowed by the memory model (not just fences). This technique completes writes speculatively and, therefore, needs to buffer speculative state. Depending on the design, the speculative state is stored in large purposely-built post-retirement buffers or in L1 caches modified with speculative read/write bits. In some cases, the speculation is performed in chunks of instructions. None of our schemes completes writes speculatively, not even *W+*.

Our work is also related to schemes that enforce SC or identify SC violations. Examples are Conflict Ordering (CO) [21], End-to-End SC [22, 31], Vulcan [24], and Volition [25]. These schemes are concerned about the reordering of all accesses; in Asymmetric fences, we are concerned only about the reordering of the accesses across fences.

Software researchers have built on a cycle-detection algorithm [29] to insert fences in codes to guarantee SC (e.g., [7, 18, 33]). Their goal is to minimize the number of fences added to guarantee SC. Our work is complementary, as we help them minimize the overhead of SC guarantees.

In C/C++ and Java, it is possible to avoid exposing fences and, instead, provide implicit ordering with respect to a single update. This is supported by the ARMv8 [2] and Itanium [12] load-acquire and store-release instructions. Adapting *wF*s to these environments is interesting future work.

9. Conclusions

Past fence proposals improved performance by allowing post-fence accesses to complete before the fence completes. Unfortunately, such proposals present implementation challenges caused by requiring global state and structures.

The goal of this paper was to improve both the performance and the implementability of fences. We used a fence design like the most aggressive ones but without the global state (*Weak Fence* or *wF*) combined with a conventional fence (*Strong Fence* or *sF*) for the less performance-critical threads. We called the result an Asymmetric fence group. We proposed a taxonomy of Asymmetric fence groups. Compared to past aggressive fences, Asymmetric fences both are substantially easier to implement and have higher average performance. Hence, they offer the best performance-cost tradeoff that we are aware of. The two main designs presented (*WS+* and *W+*) speed-up workloads under TSO by an average of 13% and 21%, respectively, over conventional fences. The designs require different hardware: *WS+* requires the Order bit and operation, while *W+* requires checkpointing, triggering a timeout when a cycle is suspected, and rolling back.

Acknowledgments

We thank Hans Boehm and the anonymous reviewers.

References

[1] Rochester Software Transactional Memory. http://www.cs.rochester.edu/research/synchronization/rstm/.

[2] ARM. ARMv8-A Reference Manual, Issue A.d. http://infocenter.arm.com.

[3] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *International Symposium on Computer Architecture*, June 2009.

[4] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[5] Dave Dice, Mark Moir, and William Scherer. Quickly Reacquirable Locks. *Technical Report, Sun Microsystems Inc.*, 2003.

[6] Dave Dice and Nir Shavit. TLRW: Return of the Read-write Lock. In *Symposium on Parallelism in Algorithms and Architectures*, June 2010.

[7] Yuelu Duan, Xiaobing Feng, Lei Wang, Chao Zhang, and Pen-Chung Yew. Detecting and Eliminating Potential Violations of Sequential Consistency for Concurrent C/C++ Programs. In *International Symposium on Code Generation and Optimization*, March 2009.

[8] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. WeeFence: Toward Making Fences Free in TSO. In *International Symposium on Computer Architecture*, June 2013.

[9] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Conference on Programming Language Design and Implementation*, June 1998.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *International Symposium on Computer Architecture*, June 1990.

[11] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, June 1999.

[12] Intel. Intel Itanium Architecture Software Developer's Manual, Revision 2.3. http://www.intel.com/design/itanium/ manuals/iiasdmanual.htm, May 2010.

[13] Intel Corp. *IA-32 Intel Architecture Software Developer Manual, Volume 2: Instruction Set Reference*. 2002.

[14] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock Reservation: Java Locks Can Mostly Do without Atomic Operations. In *Conference on Object-Oriented Programming, Systems, Language, and Applications*, November 2002.

[15] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-Based Memory Fences. In *Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[16] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, August 1974.

[17] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, July 1979.

[18] Jaejin Lee and D.A. Padua. Hiding Relaxed Memory Consistency with Compilers. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[19] C. Lin, V. Nagarajan, and R. Gupta. Address-aware Fences. In *International Conference on Supercomputing*, June 2013.

[20] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient Sequential Consistency using Conditional Fences. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2010.

[21] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2012.

[22] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-preserving Compiler. In *Conference on Programming Language Design and Implementation*, June 2011.

[23] Chi Cao Minh, Jaewoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *International Symposium on Workload Characterization*, September 2008.

[24] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *International Symposium on Microarchitecture*, December 2012.

[25] Xuehai Qian, Benjamin Sahelices, Josep Torrellas, and Depei Qian. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.

[26] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Symposium on Parallelism in Algorithms and Architectures*, June 1997.

[27] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 2007.

[28] Douglas C. Schmidt and Tim Harrison. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-Safe Objects. In *Conference on Pattern Languages of Programming*, 1996.

[29] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, April 1988.

[30] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Symposium on Principles of Distributed Computing*, August 1995.

[31] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd D. Millstein, and Madanlal Musuvathi. End-to-End Sequential Consistency. In *International Symposium on Computer Architecture*, June 2012.

[32] SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. 1994.

[33] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.